Metaheurísticas

Problema del Aprendizaje de Pesos en Características.

Práctica 3: Enfriamiento Simulado, Búsqueda Local Reiterada y Evolución Diferencial.

UNIVERSIDAD DE GRANADA

E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN

UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Grado en Ingeniería Informática. Tercero.

Curso 2018-2019.

Daniel Bolaños Martínez

76592621-E

danibolanos@correo.ugr.es

Grupo 1 - Martes 17:30h

Índice

1. Descripción del Problema

El problema del APC consiste en optimizar el rendimiento de un clasificador basado en vecinos más cercanos a partir de la inclusión de pesos asociados a las características del problema que modifican su valor en el momento de calcular las distancias entre ejemplos.

La variante del problema del APC que afrontaremos busca optimizar tanto la precisión como la complejidad del clasificador. Así se puede formular como:

$$Maximizar\ F(W) = \alpha \cdot tasa\_clas(W) + (1 - \alpha) \cdot tasa\_red(W)$$

Partimos de un vector de datos $T = \{t_1, \ldots, t_n\}$ donde cada dato contiene un conjunto de características o *traits* $\{x_1, \ldots, x_n\}$.

El problema se reduce en calcular un vector de pesos $W = \{w_1, \ldots, w_n\}$ donde el peso $w_i \in [0, 1]$ pondera la característica x_i. A este vector lo denominaremos vector de entrenamiento o aprendizaje y nos permitirá clasificar otro conjunto de datos desconocido al que llamaremos vector de test o validación.

Para aumentar la fiabilidad del proceso usaremos el método *5-fold cross validation* que consiste en crear cinco particiones distintas de datos repartidos equitativamente según su clase. Las particiones las dividiremos de tal forma que dediquemos una a *test* y cuatro a *train*.

Para clasificar los datos, implementamos el algoritmo **k-NN** en su versión con $k = 1$. Este algoritmo asigna a cada dato la clase su vecino más cercano calculado a partir de la *distancia euclídea*.

Calcularemos el vector de pesos con los diferentes algoritmos y usaremos nuestro clasificador **k-NN** para valorar el rendimiento de los resultados. Además es necesario calcular los diferentes porcentajes para cada conjunto de datos valorados. Definiremos:

$$tasa\_clas = 100 \cdot \frac{n^o\ instancias\ bien\ clasificadas\ en\ T}{n^o\ instancias\ en\ T}$$

$$tasa\_red = 100 \cdot \frac{n^o\ valores\ w_i < 0.2}{n^o\ caracteristicas}$$

Para proceder a los cálculos debemos partir de unos datos normalizados. En la primera práctica, implementamos un algoritmo **Greedy (Relief)** y uno de **Búsqueda Local** y en esta hemos desarrollado cuatro algoritmos **Enfriamiento Simulado**, **Búsqueda Local Reiterada** y dos versiones de **Evolución Diferencial**. Los algoritmos son ejecutados sobre un conjunto de datos que contienen los archivos *colposcopy*, *ionosphere*, *texture* todos ellos en el formato elegido *.csv*.

2. Descripción de los Algoritmos empleados en el Problema

Nuestro esquema de representación de la solución, será un vector de pesos W que nos servirá para valorar la bondad de los datos en clasificaciones futuras. El vector será del tipo $W = \{w_1, \ldots, w_n\}$ donde cada $w_i \in [0, 1]$ y cada w_i gradúa el peso asociado a cada característica y pondera su importancia.

Nuestro algoritmo **k-NN** medirá la bondad de los pesos calculados mediante cada algoritmo, recibiendo como parámetros un vector de datos de entrenamiento, un vector de datos de test y el vector de pesos calculado por el algoritmo seleccionado. El clasificador devolverá un *struct Resultados* que contiene las tasas de clasificación y reducción calculadas a partir del número de aciertos entre ambos conjuntos, así como el tiempo que ha tardado en ejecutarse.

La tasa de clasificación se calcula como la media de los porcentajes de acierto obtenidos por cada método en cada partición del conjunto de datos. A mayor tasa de clasificación, mejor será el vector de pesos W generado por nuestro algoritmo.

La tasa de reducción corresponde al porcentaje de reducción obtenido en la selección del subconjunto de características respecto al total. Una tasa de reducción alta indica que necesitaremos menos atributos para clasificar los datos en un futuro.

El pseudocódigo del algoritmo **k-NN** es el siguiente:

```
function KNN_LOO(train, test, w)
   for i=0, i < tamaño test, i++ do
      pos = nearestNeighbour_LOO(train, test[i], w)
      if train[pos].clase = test[i].clase then
          aciertos++
      end if
   end for

   for i=0, i < tamaño w, i++ do
     if w[i] < 0.2 then
       num_w_menor++
     end if
   end for

  tasa_clas = 100.0*(aciertos / tamaño test)
  tasa_red = 100.0*(num_w_menor / tamaño w)
end function
```

Para calcular el vecino más cercano, calculamos el que tenga menor distancia euclídea respecto del que estemos valorando. Además, ha sido programado aplicando leave-one-out, ya que cuando se utiliza en la **Búsqueda Local**, no podemos calcular el mejor vecino sin tener en cuenta esto, porque estamos

aplicando el **k-NN** sobre el mismo conjunto de entrenamiento y podríamos obtener como mejor vecino el mismo dato que estamos comparando.

```
function nearestNeighbour_LOO(train, actual, w)
   mejor_distancia = n traits primera partición de train + 1

   for i=0, i < tamaño train, i++ do
      // leave-one-out
      if actual != train[i] then
         // .t accede al vector de caracteristicas
         distancia_actual = euclideanDistance(train[i].t, actual.t, w)
         if distancia_actual < mejor_distancia then
            mejor_distancia = distancia_actual
            pos = i
         end if
      end if
   end for
end function
```

Finalmente utilizamos la función *euclideanDistance* para calcular la distancia euclídea entre los diferentes vectores de características teniendo en cuenta los pesos cuyo valor supere 0.2.

```
function euclideanDistance(v1, v2, w)
   for i=0, i < tamaño v1, i++ do
      if w[i] >= 0.2 then
         dist = dist + w[i]*(v2[i]-v1[i])*(v2[i]-v1[i])
      end if
   end for
end function
```

Además, he añadido una versión de la función **k-NN** que utiliza una modalidad del *nearestNeighbour* sin el criterio de Leave-One-Out. Esta versión del algoritmo será usada cuando apliquemos el **k-NN** sobre dos conjuntos de datos diferentes.

```
function nearestNeighbour(train, actual, w)
   mejor_distancia = n traits primera partición de train + 1

   for i=0, i < tamaño train, i++ do
         // .t accede al vector de caracteristicas
         distancia_actual = euclideanDistance(train[i].t, actual.t, w)
         if distancia_actual < mejor_distancia then
            mejor_distancia = distancia_actual
            pos = i
         end if
   end for
end function
```

En las dos versiones de **Evolución Diferencial** he creado un struct *Cromosoma*

que contiene la información básica de cada elemento con el que vamos a trabajar (la lista de características y su valor de evaluación).

```
struct Cromosoma{
    vector<double> w;
    double pts;
};
```

Para seleccionar 3 y 2 padres excluyentes de la población en los algoritmos **DE/Rand** y **DE/current-to-best** respectivamente, he creado dos funciones auxiliares cuyo pseudocódigo es el siguiente:

```
function select_3_parents(generador)
  vector de 3 int index
  index[0] = random_int(generador)

  do
    index[1] = random_int(generador)
  while index[0]==index[1]

  do
    index[2] = random_int(generador)
  while index[2]==index[0] || index[2]==index[1]
  return index
end function
```

```
function select_2_parents(generador)
  vector de 2 int index
  index[0] = random_int(generador)

  do
    index[1] = random_int(generador)
  while index[0]==index[1]
  return index
end function
```

La búsqueda local utilizada en el algoritmo **ILS** es la misma de la práctica 1, salvo la modificación de que el número máximo de iteraciones, son 1000 en lugar de 15000, será el propio **ILS** el que se encargue de realizar las 15 llamadas de la **BL** para realizar como máximo las 15000 evaluaciones.

```
function BL(train, w)
   w = distribucion_uniforme(0,1)
   index = {0,...,w.size()}
   mezcla los valores de index
   // clasifica el vector de pesos w con KNN y
   // calcula su agregado como tasa de evaluación
```

```
   antiguo = KNN(train, train, w)
   agr_ant = agregado(antiguo.clas,antiguo.red)

   while iter < 1000 and neighbour < tamaño w*MAX_NEIGHBOUR do
      aux = index[iter % tamaño w]
      w_mut = w
      w_mut[aux] += normal(generator)
      truncar el vector de pesos mutado
      //clasifica el vector de pesos mutado
      // y calcula su agregado
      agr_new = agregado(resultados KNN_LOO)
      iter++
      if agr_new > agr_ant then
         w = w_mut
         agr_ant = agr_new
         neighbour = 0
      else
         neighbour++
      end if
      if iter % tamaño w = 0 then
         mezcla los valores de index
      end if
   end while
end function
```

La función agregado calcula la tasa de agregado de los resultados obtenidos al clasificar los datos de entrenamiento con el vector de pesos en cada caso y utiliza este resultado para evaluar la bondad de la solución obtenida. En este caso usamos alpha=0.5.

```
function agregado(t_clas, t_red)
  alpha*t_clas+(1.0-alpha)*t_red
end function
```

3. Pseudocódigo de los Algoritmos

3.1. Algoritmo Greedy (Relief)

El algoritmo se basa en incrementar el peso de aquellas características que mejor separan a ejemplos que son enemigos entre sí y reducir el valor del peso en aquellas características que separan ejemplos que son amigos entre sí.

El pseudocódigo del algoritmo **Greedy (Relief)** es el siguiente:

```
function Relief(train, w)
   w = {0,...,0}
```

```
   for i=0, i < tamaño train, i++ do
      nearestFriendEnemy(train, train[i], friend, enemy)
      for j=0, j < tamaño w, j++ do
         w[j]= w[j]+|train[i].t[j]-train[i].t[enemy]|
               -|train[i].t[j]-train[i].t[friend]|
      end for
   end for

   w_max = máximo del vector de pesos w

   for i=0, i < tamaño w, i++ do
      if w[i] < 0 then
         w[i] = 0
      else
         w[i] = w[i] / w_max
      end if
   end for
end function
```

La función *nearestFriendEnemy* calcula el amigo y el enemigo más cercano a un dato dado en función de la *distancia euclídea*. Un dato se considera enemigo si tiene la clase distinta y amigo si tiene la misma clase del dato actual. La función emplea leave-one-out para evitar comparar distancias con el dato actual.

```
function nearestFriendEnemy(train, actual, friend, enemy)
   for i=0, i < tamaño train, i++ do
      if actual != train[i] then
         distancia_actual = euclideanDistance(train[i].t, actual.t)
         if train[i].category = actual.category then
            if distancia_actual < mejor_distancia_a then
               mejor_distancia_a = distancia_actual
               friend = i
            end if
         end if
      else
         if distancia_actual < mejor_distancia_e then
            mejor_distancia_e = distancia_actual
            enemy = i
         end if
      end if
   end for
end function
```

3.2. Enfriamiento Simulado (ES)

Para este algoritmo, se empleará el esquema de enfriamiento de Cauchy, que define la temperatura y el valor de β de la siguiente forma:

$$T_{k+1} = \frac{T_k}{1 + \beta \cdot T_k} \qquad \beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

Donde T_0 es la temperatura inicial, T_f la final y M el número de enfriamientos a realizar.

Para poder generar soluciones nuevas, deberemos modificar/mutar el vector W añadiendo a cada elemento un valor que siga una distribución normal de media 0 y varianza $\sigma = 0.3$, pero este método puede proporcionar soluciones negativas, por lo que debemos de truncar los valores negativos a 0.

Tendremos un bucle interno para la condición de enfriamiento y uno externo para la condición de parada.

La condición de enfriamiento finalizará la iteración actual bien cuando se haya generado un número máximo de vecinos o un número máximo de éxitos.

La condición de parada hará finalizar el algoritmo cuando se llegue a las 15000 iteraciones o bien cuando el número de éxitos del enfriamiento actual sea 0.

La temperatura inicial tomará el valor siguiente:

$$T_0 = \frac{\mu \cdot C(S_0)}{-ln(\phi)}$$

Donde $\mu = \phi = 0.3$ y $T_f = 10^{-3} < T_0$ siempre.

El pseudocódigo del algoritmo **ES** es el siguiente:

```
function ES(train, w)
  w = distribucion_uniforme(0,1)
  //inicializamos los valores
  max_vecinos = 10*tamaño w
  max_exitos = 0.1*max_vecinos
  M = 15000 / max_vecinos
  num_exitos = max_exitos

  // Evaluamos la solución inicial
  agr_ant = agregado(resultados KNN_LOO)
  num_eval++
  w_mejor = w
  agr_mejor = agr_ant

  // MU = PHI = 0.3 en las ejecuciones
```

```
  To = (MU*agr_mejor)/(-log(PHI))
  Tf = 0.001
  beta = (To-Tf)/(M*To*Tf)
  Tk = To

  //Comprobamos que Tf es menor que To
  //Si no se cumpliese multiplicamos por 0.1 la final
  while Tf > To do
    Tf = Tf*0.1
  end while

  while num_eval < 15000 && num_exitos != 0 do
    num_exitos = 0
    neighbour = 0
    while num_eval < 15000 && num_exitos < max_exitos && neighbour < max_vecinos do
      mutar sobre una característica aleatoria de w
      truncar el resultado
      // Calculo la tasa de agregado del vector mutado
      agr_new = agregado(resultados KNN_LOO)
      num_eval++
      neighbour++
      // Cálculo del incremento de la diff
      diff = agr_ant-agr_new
      if diff < 0 || U(0,1) <= exp(-diff/Tk) then
        w = w_mut
        agr_ant = agr_new
        num_exitos++
        if agr_new > agr_mejor then
          w_mejor = w_mut
          agr_mejor = agr_new
        end if
      end if
    end while
    // Enfriamiento (Esquema de Cauchy)
    Tk = Tk/(1.0+beta*Tk)
  end while

  w = w_mejor
end function
```

3.3. Búsqueda Local Reiterada (ILS)

Este algoritmo consiste en generar una solución aleatoria y aplicar de forma reiterada **Búsqueda Local** sobre ella.

Cada vez que apliquemos una **BL** estudiaremos si la solución obtenida es mejor o peor que la mejor anterior y se realizará una mutación sobre la mejor de las dos.

La mutación diferirá del resto de mutaciones realizadas en algoritmos ya que usaremos una distribución normal con $\sigma = 0.4$ en vez del 0.3 usual. Realizaremos 15 iteraciones de **BL** y se aplicará la mutación sobre el 10% de las características del vector de pesos.

El pseudocódigo del algoritmo **ILS** es el siguiente:

```
function ILS(train, w)
  w = distribucion_uniforme(0,1)
  num_mutaciones = ceil(0.1*tamaño w)
  // Aplicamos la BL y tomamos el agr actual como mejor
  BL(train, w)
  agr_mejor = agregado(resultados KNN_LOO)
  w_mejor = w

  // Hacemos 14 iteraciones
  for i=1, i < 15, i++ do
    // Mutamos la mejor solución
    for j=0, j < num_mutaciones, j++ do
      mutar y truncar una característica aleatoria
    end for
    BL(train, w)
    agr_new = agregado(resultados KNN_LOO)
    // Si la nueva solución es mejor se toma esa
    if agr_new > agr_mejor then
      w_mejor = w
      agr_mejor = agr_new
    end if
    w = w_mejor
  end for
end function
```

3.4. Evolución Diferencial (DE)

La **Evolución Diferencial** es un modelo evolutivo propuesto para la optimización con parámetros reales que enfatiza la mutación y utiliza un operador de cruce a posteriori.

Para esta práctica se piden dos versiones del algoritmo, los cuales tienen en común el proceso de Recombinación binomial y sustitución de la población con los mejores hijos generados.

$$u_{j,i,g} = \begin{cases} v_{j,i,g} & if\ rand_j[0,1] \leq CR\ or\ j = j_{rand} \\ x_{j,i,g} & otherwise \end{cases}$$

La diferencia entre ambas versiones aparece en el proceso de mutación:

- DE/rand/1:
$$V_{i,G} = X_{r_1,G} + F \cdot (X_{r_2,G} - X_{r_3,G})$$
- DE/current-to-best/1:
$$V_{i,G} = X_{i,G} + F \cdot (X_{best,G} - X_{i,G}) + F \cdot (X_{r_1,G} - X_{r_2,G})$$

Donde r_i son los padres i-ésimos seleccionados de forma aleatoria excluyente de la población y X_{best} denota el mejor vector de la generación G. El tamaño de la población será de 50 vectores. Y tomaremos las constantes $CR = F = 0.5$.

3.4.1. DE/rand/1

```
function DE_rand(train, w)
  w = distribucion_uniforme(0,1)
  // Inicializamos la población inicial y guardamos la pos del mejor
  for i=0, i < 50, i++ do
    pbl[i].pts = agregado(resultados KNN_LOO)
    if pbl[i].pts > pbl[mejor].pts then
      mejor = i
    end if
    num_eval++
  end for

  // Condición de parada
  while num_eval < 15000 do
    for i=0, i < 50, i++ do
      // Generamos los 3 padres aleatorios excluyentes de la población
      padres = select_3_parents(generador)

      // Seleccionamos una caracteristica random
      random = random_int_mut(generador)

      for j=0, j < tamaño w, j++ do
        // Recombinación binomial
        if U(0,1) < 0.5 || random == j then
          x_pesos[j] = padre1.w[j]+0.5*(padre2.w[j]-padre3.w[j])
          trucar x_pesos[j]
        else
          x_pesos[j] = pbl[i].w[j]
```

```
        end if
      end for
      offspring[i].w = x_pesos
    end for

    // Evaluamos los nuevos hijos
    for i=0, i < 50, i++ do
      offspring[i].pts = agregado(resultados KNN_LOO)
      num_eval++
    end for

    // Sustituyo los hijos mejores en la población
    for i=0, i < 50, i++ do
      if offspring[i].pts > pbl[i].pts then
        pbl[i] = offspring[i]
      end if
    end for

    //Obtengo el mejor de la nueva poblacion
    for i=0, i < 50, i++ do
      if pbl[i].pts > pbl[mejor].pts then
        mejor = i
      end if
    end for
  end while

  w = pbl[mejor].w
end function
```

3.4.2. DE/current-to-best/1

```
function DE_best(train, w)
  w = distribucion_uniforme(0,1)
  // Inicializamos la población inicial y guardamos la pos del mejor
  for i=0, i < 50, i++ do
    pbl[i].pts = agregado(resultados KNN_LOO)
    if pbl[i].pts > pbl[mejor].pts then
      mejor = i
    end if
    num_eval++
  end for

  // Condición de parada
  while num_eval < 15000 do
    for i=0, i < 50, i++ do
```

```
      // Generamos los 2 padres aleatorios excluyentes de la población
      padres = select_2_parents(generador)

      // Seleccionamos una caracteristica random
      random = random_int_mut(generador)

      for j=0, j < tamaño w, j++ do
        // Recombinación binomial
        if U(0,1) < 0.5 || random == j then
          x_pesos[j] = pbl[i].w[j]+0.5*(pbl[mejor].w[j]-pbl[i].w[j])
          +0.5*(padre1.w[j]-padre2.w[j])
          trucar x_pesos[j]
        else
          x_pesos[j] = pbl[i].w[j]
        end if
      end for
      offspring[i].w = x_pesos
    end for

    // Evaluamos los nuevos hijos
    for i=0, i < 50, i++ do
      offspring[i].pts = agregado(resultados KNN_LOO)
      num_eval++
    end for

    // Sustituyo los hijos mejores en la población
    for i=0, i < 50, i++ do
      if offspring[i].pts > pbl[i].pts then
        pbl[i] = offspring[i]
      end if
    end for

    //Obtengo el mejor de la nueva poblacion
    for i=0, i < 50, i++ do
      if pbl[i].pts > pbl[mejor].pts then
        mejor = i
      end if
    end for
  end while

  w = pbl[mejor].w
end function
```

4. Descripción en Pseudocódigo de los Algoritmos de Comparación

El proceso para comparar los datos obtenidos para cada algoritmo es siempre el mismo y se realiza en la función *ejecutar* para cada algoritmo programado.

Primero obtenemos los pesos usando el algoritmo a comparar y seguidamente clasificamos con el **k-NN** los datos de train y test sobre ese vector de pesos obtenido. Por último devolvemos en un *struct Resultados* las tasas de clase, reducción, agregado y el tiempo que ha tardado en ejecutar y repetimos el proceso cambiando el índice de la partición de test.

```
function ejecutarAlgoritmo(particion, i)
   test = toma la partición i
   train = toma la suma de datos de las particiones !=
   w = Algoritmo(train, w)
   resultados = KNN(train, test, w)
end function
```

5. Procedimiento del Desarrollo de la Práctica y Manual de Usuario

La práctica ha sido programada en el lenguaje C++. Para el desarrollo de la práctica, primero he necesitado leer los archivos que nos proporcionan los datos, para ello, he optado por pasar los archivos .arff a .csv, un formato que al menos para mí, es más manipulable. Seguidamente y con ayuda de la función *read_csv*, guardo los datos en memoria en un vector de estructuras que he denominado *FicheroCSV*. Cada *FicheroCSV* contiene la información de cada línea del fichero *csv* original incluyendo un vector de los datos y un string que indica la clase de los mismos. Por tanto, consideraremos cada conjunto de datos como un *vector<FicheroCSV>*.

Seguidamente, y para aplicar la *5-fold cross validation*, creo un vector con 5 particiones que contienen punteros (para reducir el coste de memoria) a estructuras *FicheroCSV* repartidas de forma equitativa. Los datos se introducen en la partición ya normalizados.

Una vez que tenemos el vector<vector<FicheroCSV*>>, ya podemos repartir los datos en train y test y proceder como hemos descrito anteriormente con los algoritmos programados.

Para la primera práctica, desarrollé una versión del **k-NN** sin leave-one-out, seguidamente programé el algoritmo **Greedy (Relief)** y finalmente la **Búsqueda Local** con la cual tuve que modificar parte del código del **k-NN** para que aplicase el leave-one-out.

Para la tercera práctica, programé los algoritmos en este orden **Enfriamiento**

Simulado, **Búsqueda Local Reiterada** (usando la versión de **BL** de la práctica 1) y **Evolución Diferencial** (versión rand y current-to-best).

En el archivo *LEEME.txt* se encuentra explicado el proceso para replicar las ejecuciones del programa. Así que me limitaré a resumirlo:

Existe un make que genera el ejecutable del programa. El programa ejecutable ha sido compilado con g++ y optimizado con -O2 para reducir los tiempos de ejecución.

./bin/p3 ./data/archivo.csv seed

Ejecuta los seis algoritmos programados en ambas prácticas sobre el conjunto de datos contenido en el *archivo.csv* (con $archivo = \{colposcopy, ionosphere, texture\}$) usando como semilla el número *seed* pasado como parámetro en el **ES**, **ILS** y **DE**.

./bin/p3 seed

Crea un archivo *tablas_seed.csv* con los resultados de ejecutar los tres algoritmos implementados sobre los tres conjuntos de datos (*colposcopy.csv*, *ionosphere.csv*, *texture.csv*) y vuelca su contenido en una tabla. El parámetro *seed* indica la iniciacización de la semilla.

Tanto los ficheros .csv como el fichero tablas_seed.csv que se genere se podrán encontrar en el directorio ./data.

6. Experimentos y Análisis de Resultados

Para la obtención de los resultados de esta práctica, hemos utilizado los siguientes conjuntos de datos:

- **Colposcopy**: Conjunto de datos de secuencias colposcópicas extraídas a partir de las características de las imágenes. Contiene 287 ejemplos con 62 características cada uno clasificados en 2 categorías.
- **Ionosphere**: Conjunto de datos de radar que indican el número de electrones libres en la ionosfera partir de las señales procesadas. Contiene 352 ejemplos con 34 características cada uno clasificados en 2 categorías.
- **Texture**: Conjunto de datos para distinguir entre 11 tipos de texturas diferentes a partir de imágenes. Contiene 550 ejemplos con 40 características cada uno clasificados en 11 categorías.

Los algoritmos de **ES**, **ILS** y **DE** dependen de un parámetro que especifica la semilla para le generación de números aleatorios. Voy a utilizar para el análisis de estos resultados SEED=3. Los resultados se guardarán por tanto en el archivo *tablas_3.csv* del directorio *data*.

Para la generación de vecinos y mutación se van a usar los datos generados por una distribución normal de media 0 y varianza $\sigma = 0.3$ (a excepción de **ILS**

donde es 0.4). Como criterio de parada en la **Búsqueda Local** utilizada en **ILS**, se va a usar el número de evaluaciones que se han realizado así como el número vecinos por característica explorados. En nuestro caso los valores serán 1000 y $20 \cdot tam\_vector\_pesos$ respectivamente.

Las tablas se generan automáticamente con la ejecución del programa y contienen los resultados para cada partición, así como la media de los resultados para cada algoritmo.

A continuación, se muestran las tablas para cada fichero y cada algoritmo utilizado:

	COLPOSCOPY		IONOSPHERE		TEXTURE	
	%tasa_clas	%tasa_red	%tasa_clas	%tasa_red	%tasa_clas	%tasa_red
1-NN	74,9062	0	86,5996	0	92,5455	0
Relief	74,9062	40,9677	87,4567	2,94118	93,0909	5,5
ES	71,4398	85,1613	87,4527	87,0588	88,9091	85,5
ILS	73,533	76,4516	89,4487	87,6471	89,4545	84
DE/Rand/1	67,2353	93,5484	86,3179	92,3529	90,1818	87,5
DE/current-to-best/1	74,8639	72,2581	87,167	77,6471	89,6364	77,5

Figure 1: Tabla datos comparativa

Teniendo en cuenta solo los algoritmos de la primera práctica, podemos ver que los mejores resultados son obtenidos por el algoritmo de **Greedy-Relief**, igualando en tasa de clasificación al **1-NN** para el conjunto de datos *colposcopy*.

Respecto a la tercera práctica, las tasas de clasificación de los diferentes algoritmos son similares en media, distan solo en 5% como mucho. Siendo el algoritmo **Greedy-Relief** el que obtiene mejor tasa de media.

Si observamos la tasa de reducción, en este caso, podemos ver que el que mayor tasa obtiene con diferencia es el **DE/rand/1** obteniendo una tasa cercana al 95%. Seguido muy de cerca por el **Enfriamiento Simulado**.

A diferencia de prácticas anteriores podemos ver que aunque varíen los conjuntos de datos y por ello la cantidad de los mismos, el algoritmo **DE/rand/1** siempre obtiene la mayor tasa de clasificación, por lo que podemos decir que se adapta bastante bien tanto en la clasificación de nuevos datos *test* como en la cantidad y tipo de los mismos.

Podemos ver los resultados totales obtenidos para cada conjunto de datos:

	COLPOSCOPY			
	1-NN			
	%tasa_clas	%tasa_red	Agregado	T(seg)
Partición 1	81,0345	0	40,5172	0,003552
Partición 2	70,6897	0	35,3448	0,001941
Partición 3	77,193	0	38,5965	0,00149
Partición 4	68,4211	0	34,2105	0,00115
Partición 5	77,193	0	38,5965	0,001104
Media:	**74,9062**	**0**	**37,4531**	**0,0018474**
	Greedy-Relief			
Partición 1	79,3103	37,0968	58,2036	0,004727
Partición 2	72,4138	35,4839	53,9488	0,004518
Partición 3	80,7018	35,4839	58,0928	0,004508
Partición 4	70,1754	38,7097	54,4426	0,004542
Partición 5	71,9298	58,0645	64,9972	0,004436
Media:	**74,9062**	**40,9677**	**57,937**	**0,0045462**
	ES			
Partición 1	72,4138	82,2581	77,3359	36,8027
Partición 2	67,2414	88,7097	77,9755	36,99
Partición 3	77,193	80,6452	78,9191	35,832
Partición 4	63,1579	87,0968	75,1273	36,1956
Partición 5	77,193	87,0968	82,1449	36,2015
Media:	**71,4398**	**85,1613**	**78,3005**	**36,4044**

Figure 2: Algoritmos 1-NN, Relief y ES en Colposcopy.csv

	COLPOSCOPY			
	%tasa_clas	%tasa_red	Agregado	T(seg)
	ILS			
Partición 1	70,6897	77,4194	74,0545	40,9106
Partición 2	72,4138	75,8065	74,1101	39,833
Partición 3	75,4386	77,4194	76,429	40,2097
Partición 4	70,1754	75,8065	72,9909	40,211
Partición 5	78,9474	75,8065	77,3769	39,3223
Media:	**73,533**	**76,4516**	**74,9923**	**40,0973**
	DE/Rand/1			
Partición 1	81,0345	91,9355	86,485	41,6943
Partición 2	56,8966	93,5484	75,2225	38,6434
Partición 3	63,1579	93,5484	78,3531	38,9677
Partición 4	70,1754	95,1613	82,6684	41,7546
Partición 5	64,9123	93,5484	79,2303	41,7289
Media:	**67,2353**	**93,5484**	**80,3919**	**40,5578**
	DE/current-to-best/1			
Partición 1	79,3103	72,5806	75,9455	37,6754
Partición 2	84,4828	70,9677	77,7253	38,1668
Partición 3	78,9474	74,1935	76,5705	37,9306
Partición 4	61,4035	74,1935	67,7985	38,8854
Partición 5	70,1754	69,3548	69,7651	39,2805
Media:	**74,8639**	**72,2581**	**73,561**	**38,3877**

Figure 3: Algoritmos ILS y DE en Colposcopy.csv

	IONOSPHERE			
	1-NN			
	%tasa_clas	%tasa_red	Agregado	T(seg)
Partición 1	90,1408	0	45,0704	0,002019
Partición 2	80	0	40	0,001399
Partición 3	82,8571	0	41,4286	0,001052
Partición 4	92,8571	0	46,4286	0,000906
Partición 5	87,1429	0	43,5714	0,000898
Media:	**86,5996**	**0**	**43,2998**	**0,0012548**
	Greedy-Relief			
Partición 1	90,1408	2,94118	46,541	0,003929
Partición 2	81,4286	2,94118	42,1849	0,004313
Partición 3	82,8571	2,94118	42,8992	0,003928
Partición 4	92,8571	2,94118	47,8992	0,003958
Partición 5	90	2,94118	46,4706	0,004275
Media:	**87,4567**	**2,94118**	**45,199**	**0,0040806**
	ES			
Partición 1	91,5493	85,2941	88,4217	31,4861
Partición 2	84,2857	88,2353	86,2605	31,6211
Partición 3	87,1429	85,2941	86,2185	31,0758
Partición 4	90	88,2353	89,1176	31,1752
Partición 5	84,2857	88,2353	86,2605	30,8504
Media:	**87,4527**	**87,0588**	**87,2558**	**31,2417**

Figure 4: Algoritmos 1-NN, Relief y ES en Ionosphere.csv

	IONOSPHERE			
	%tasa_clas	%tasa_red	Agregado	T(seg)
	ILS			
Partición 1	92,9577	88,2353	90,5965	33,6726
Partición 2	84,2857	88,2353	86,2605	33,4233
Partición 3	84,2857	88,2353	86,2605	33,2096
Partición 4	92,8571	85,2941	89,0756	33,2987
Partición 5	92,8571	88,2353	90,5462	33,2148
Media:	**89,4487**	**87,6471**	**88,5479**	**33,3638**
	DE/Rand/1			
Partición 1	88,7324	91,1765	89,9544	33,2815
Partición 2	84,2857	91,1765	87,7311	33,6817
Partición 3	81,4286	94,1176	87,7731	36,9879
Partición 4	94,2857	94,1176	94,2017	35,8044
Partición 5	82,8571	91,1765	87,0168	34,9276
Media:	**86,3179**	**92,3529**	**89,3354**	**34,9366**
	DE/current-to-best/1			
Partición 1	91,5493	67,6471	79,5982	32,9646
Partición 2	84,2857	67,6471	75,9664	34,8261
Partición 3	82,8571	85,2941	84,0756	31,9895
Partición 4	88,5714	85,2941	86,9328	31,7772
Partición 5	88,5714	82,3529	85,4622	32,3362
Media:	**87,167**	**77,6471**	**82,407**	**32,7787**

Figure 5: Algoritmos ILS y DE en Ionosphere.csv

	TEXTURE			
	1-NN			
	%tasa_clas	%tasa_red	Agregado	T(seg)
Partición 1	93,6364	0	46,8182	0,005242
Partición 2	89,0909	0	44,5455	0,004109
Partición 3	94,5455	0	47,2727	0,003104
Partición 4	92,7273	0	46,3636	0,002719
Partición 5	92,7273	0	46,3636	0,002923
Media:	**92,5455**	**0**	**46,2727**	**0,0036194**
	Greedy-Relief			
Partición 1	91,8182	15	53,4091	0,011238
Partición 2	91,8182	2,5	47,1591	0,012252
Partición 3	95,4545	2,5	48,9773	0,012533
Partición 4	92,7273	2,5	47,6136	0,011493
Partición 5	93,6364	5	49,3182	0,01229
Media:	**93,0909**	**5,5**	**49,2955**	**0,0119612**
	ES			
Partición 1	87,2727	87,5	87,3864	90,5979
Partición 2	89,0909	85	87,0455	89,2717
Partición 3	94,5455	85	89,7727	87,8699
Partición 4	87,2727	85	86,1364	88,0044
Partición 5	86,3636	85	85,6818	87,6769
Media:	**88,9091**	**85,5**	**87,2045**	**88,6842**

Figure 6: Algoritmos 1-NN, Relief y ES en Texture.csv

	TEXTURE			
	%tasa_clas	%tasa_red	Agregado	T(seg)
	ILS			
Partición 1	89,0909	85	87,0455	93,9756
Partición 2	93,6364	85	89,3182	93,955
Partición 3	93,6364	82,5	88,0682	93,5651
Partición 4	84,5455	85	84,7727	94,0525
Partición 5	86,3636	82,5	84,4318	94,0194
Media:	**89,4545**	**84**	**86,7273**	**93,9135**
	DE/Rand/1			
Partición 1	90,9091	87,5	89,2045	89,0088
Partición 2	90,9091	87,5	89,2045	95,5129
Partición 3	91,8182	87,5	89,6591	93,2198
Partición 4	87,2727	87,5	87,3864	89,4205
Partición 5	90	87,5	88,75	88,738
Media:	**90,1818**	**87,5**	**88,8409**	**91,18**
	DE/current-to-best/1			
Partición 1	86,3636	77,5	81,9318	89,2104
Partición 2	90	80	85	89,7903
Partición 3	91,8182	75	83,4091	90,3081
Partición 4	90	77,5	83,75	90,7171
Partición 5	90	77,5	83,75	92,4911
Media:	**89,6364**	**77,5**	**83,5682**	**90,5034**

Figure 7: Algoritmos ILS y DE en Texture.csv

Vemos que en general, de media todos los algoritmos programados en esta práctica, tardan tiempos similares en ejecutar los datos. A mayor cantidad de datos, más tardan, por lo que la ejecución sobre el conjunto de datos *texture.csv* es la que más tardan en media.

6.1. Sobreaprendizaje

Decimos que un algoritmo ha sido sobreentrenado, si el algoritmo empleado sobre un conjunto de datos es mejor sobre los datos de entrenamiento que sobre los de prueba. Esto es malo ya que el algoritmo puede ser bueno para un conjunto de datos específico pero perder generalidad para datos fuera del conjunto de entrenamiento.

Para ver que algoritmos sufren de este problema y por tanto son peores a la larga, primero entrenaremos un conjunto de datos y más tarde compararemos el resultado de evaluar ese conjunto de datos sobre él mismo con el de evaluarlo sobre otro conjunto de prueba.

Solo tenemos que hacer la siguiente modificación en las funciones de *ejecutarAlgoritmo*.

```
function ejecutarAlgoritmo(particion, num_part, SEED)
  ...
  start_timers();
  AGE_CA(train, w);
  results = KNN_LOO(train, train, w)
  results.tiempo = elapsed_time(REAL)
  ...
  return results
end function
```

Para no abusar del número de tablas incluidas, haremos las comparaciones sobre la tabla global de ambas ejecuciones.

	COLPOSCOPY		IONOSPHERE		TEXTURE	
	%tasa_clas	%tasa_red	%tasa_clas	%tasa_red	%tasa_clas	%tasa_red
1-NN	75,6085	0	86,7517	0	92,5455	0
Relief	78,6588	40,9677	88,0336	2,94118	94,1364	5,5
ES	84,8426	83,2258	92,7364	90	93,5	86
ILS	80,6615	77,0968	92,8053	87,6471	93,3636	85,5
DE/Rand/1	83,1027	93,5484	92,3798	92,3529	94,2727	87,5
DE/current-to-best/1	79,7885	72,2581	91,0971	77,6471	92,3636	77,5

Figure 8: Tabla datos comparativa train-train

	COLPOSCOPY		IONOSPHERE		TEXTURE	
	%tasa_clas	%tasa_red	%tasa_clas	%tasa_red	%tasa_clas	%tasa_red
1-NN	74,9062	0	86,5996	0	92,5455	0
Relief	74,9062	40,9677	87,4567	2,94118	93,0909	5,5
ES	71,4398	85,1613	87,4527	87,0588	88,9091	85,5
ILS	73,533	76,4516	89,4487	87,6471	89,4545	84
DE/Rand/1	67,2353	93,5484	86,3179	92,3529	90,1818	87,5
DE/current-to-best/1	74,8639	72,2581	87,167	77,6471	89,6364	77,5

Figure 9: Tabla datos comparativa train-test

Podemos ver que las tasas de las ejecuciones train-train son en media más altas que las de train-test, aunque no distan tanto como en el estudio realizado en la práctica anterior sobre los algoritmos **Genéticos** y **Meméticos**.

Además, podemos notar que el **DE/rand/1** es el que más sobreentrena los datos, ya que obtiene muy buenas tasas de clasificación sobre el conjunto train y pierde en calidad a la hora de clasificar los del conjunto test.

6.2. Convergencia

Finalmente haré un análisis de la convergencia de la tasa de agregación en función del número de evaluaciones para las dos variantes del algoritmo de **Evolución Diferencial**. Hago este análisis tras darme cuenta de que a pesar de seguir esquemas muy similares, un simple cambio como es el proceso de mutación puede hacer que un algoritmo consiga las mejores tasas mientras que el otro sea mucho peor.

Para realizar este estudio, he añadido en el código de ambos algoritmos las siguientes líneas de código:

```
if(num_eval % 1000 == 0)
  cout << num_eval << "" << poblacion[mejor].pts << endl;
```

De esta forma obtengo cada 1000 evaluaciones del algoritmo la puntuación (tasa de agregado) del mejor vector de la población.

Para evitar realizar una tabla extensa con los valores para cada partición, he simplificado los datos haciendo la media de los valores y he obtenido los siguientes resultados:

	DE/Rand/1		
ITER.	COLPOSCOPY	IONOSPHERE	TEXTURE
1000	67,06878	80,38882	78,84092
2000	74,2873	87,56242	85,68182
3000	78,92864	90,70922	88,88634
4000	83,03308	91,35922	89,6818
5000	84,80806	91,68932	90,13638
6000	86,2292	91,90284	90,43182
7000	86,88792	92,00972	90,5
8000	87,46748	92,0809	90,72728
9000	87,85934	92,18804	90,79544
10000	88,02084	92,36636	90,79544
11000	88,15128	92,36636	90,8409
12000	88,23824	92,36636	90,8409
13000	88,23824	92,36636	90,88636
14000	88,23824	92,36636	90,88636
15000	88,32558	92,36636	90,88636

Figure 10: Tabla datos convergencia media para DE/rand/1

	DE/current-to-best/1		
ITER.	COLPOSCOPY	IONOSPHERE	TEXTURE
1000	63,59808	72,32312	76,20454
2000	71,15156	81,01288	82,04546
3000	74,20502	83,38296	83,72728
4000	74,92528	83,94504	84,47726
5000	75,45264	84,01622	84,56818
6000	75,58308	84,01622	84,61362
7000	75,58308	84,30092	84,86362
8000	75,70072	84,30092	84,90908
9000	75,74438	84,30092	84,90908
10000	75,77466	84,30092	84,90908
11000	76,0233	84,30092	84,9318
12000	76,0233	84,30092	84,9318
13000	76,0233	84,3365	84,9318
14000	76,0233	84,37208	84,9318
15000	76,0233	84,37208	84,9318

Figure 11: Tabla datos convergencia media para DE/current-to-best/1

Además he realizado un gráfico comparativo para cada conjunto de datos de la convergencia de la tasa de agregado para cada conjunto de datos comparando cada versión del algoritmo:

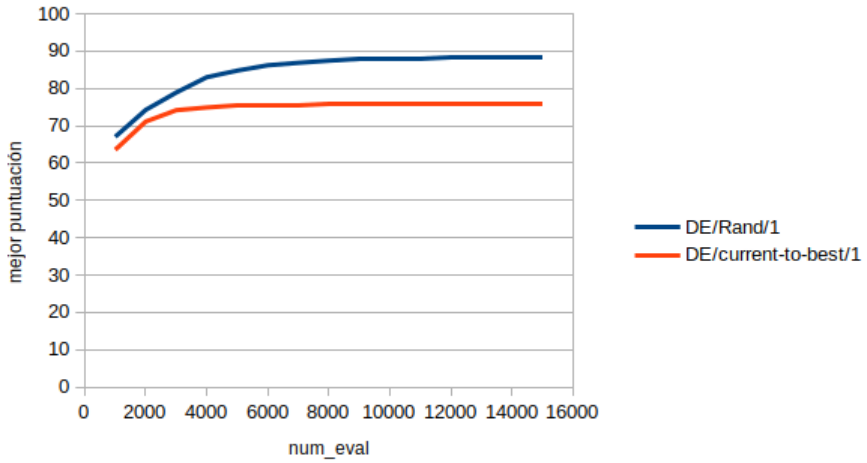

Figure 12: Gráfica convergencia en Colposcopy.csv

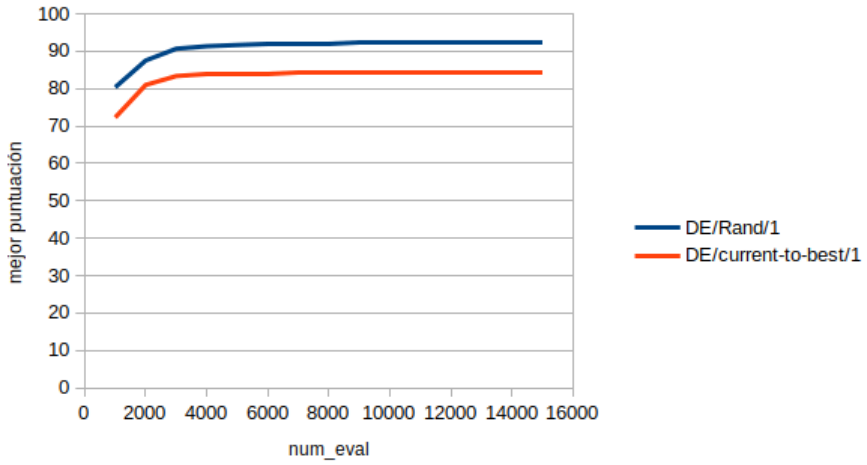

Figure 13: Gráfica convergencia en Ionosphere.csv

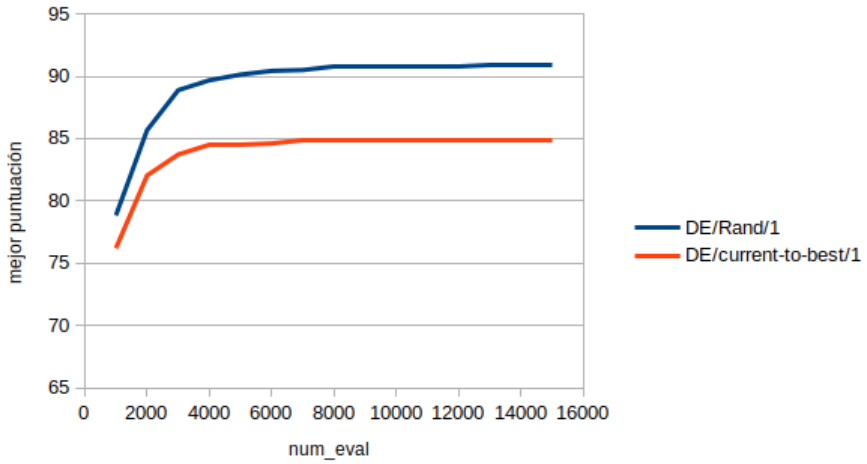

Figure 14: Gráfica convergencia en Texture.csv

En general, podemos ver que en ambos algoritmos los resultados convergen bastante rápido, desaprovechando así iteraciones. También podemos observar que **DE/rand/1** converge más lento que **DE/current-to-best/1** esto es debido a que elegir padres aleatorios ralentiza su convergencia, haciendo que sea mucho mejor al aumentar el número de iteraciones.

Podemos ver que a mayor cantidad de datos (*texture.csv*) obtenemos una convergencia más lenta y por tanto obtenemos mejores resultados para los algoritmos evolutivos **DE** en general.

7. Bibliografía

[1] Guión de Prácticas y Seminarios de la Asignatura de Metaheurísticas.